In Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, November 2001. Invited paper.

Mapping DSP Applications onto Self-timed Multiprocessors

S. S. Bhattacharyya, N. Bambha, M. Khandelia, and *V. Kianzad*
Dept. of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, USA
{ssb, nbambha, mukulk, vida}@eng.umd.edu

Abstract

Self-timed scheduling is an attractive implementation style for multiprocessor DSP systems due to its ability to exploit predictability in application behavior, its avoidance of over-constrained synchronization, and its simplified clocking requirements. However, analysis and optimization of self-timed systems under real-time constraints is challenging due to the complex, irregular dynamics of self-timed operation. This paper examines a number of intermediate representations for compiling dataflow programs onto self-timed DSP platforms, and discusses efficient techniques that operate on these representations to streamline scheduling, communication synthesis, and power management of self-timed implementations.

1 Background

Multiprocessor implementation of DSP applications involves the interaction of several complex factors including scheduling, interprocessor communication, synchronization, iterative execution, and more recently, voltage scaling for low power implementation. Addressing any one of these factors in isolation is itself typically intractable in any optimal sense; at the same time, with the increasing trend toward multi-objective implementation criteria in the synthesis of embedded software, it is desirable to understand the joint impact of these factors. In this paper, we examine several high-level, intermediate representations that have been developed to analyze and optimize various multiprocessor DSP implementation factors and manage their interactions.

The techniques discussed in this paper pertain to system specifications based on iterative synchronous dataflow (*SDF*) graphs [9]. Iterative SDF programming of DSP applications has been researched widely in the context of multiprocessor implementation, and numerous commercial DSP tools have been developed that incorporate SDF semantics. Examples of such tools include SPW by Cadence, COSSAP by Synopsys, and ADS by Hewlett-Packard.

In SDF, an application is represented as a directed graph in which vertices (***actors***) represent computational tasks, edges specify data dependences, and the numbers of data values (***tokens***) produced and consumed by each actor is fixed. Delays on SDF edges represent initial tokens, and specify dependencies between iterations of the actors in iterative execution. For example, if tokens produced by the k th invocation of actor A are consumed by the $(k+2)$ th invocation of actor B, then the edge (A, B) contains two delays. Actors can be of arbitrary complexity. In DSP design environments, they typically range in complexity from basic operations such as addition or subtraction to signal processing subsystems such as FFT units and adaptive filters. We refer to an SDF representation of an applications an ***application graph***.

In this paper, we use a form of SDF called *homogeneous* SDF (HSDF) that is suitable for dataflow-based multiprocessor design tools. In HSDF, each actor transfers a single token to/from each incident edge. General techniques for converting SDF graphs into HSDF are developed in [9]. We refer to a homogeneous SDF graph as a ***dataflow graph*** (***DFG***). We represent a DFG by an ordered pair (V, E), where V is the set of actors and E is the set of edges. We refer to the source and sink actors of a DFG edge e by $src(e)$ and $snk(e)$, we denote the delay on e by $delay(e)$, and we frequently represent e by the ordered pair $(src(e), snk(e))$. We say that e is an ***output edge*** of $src(e)$; e is an ***input edge*** of $snk(e)$; and e is ***delayless*** if $delay(e) = 0$. The execution time or estimated execution time of an actor v is denoted $t(v)$.

Mapping an application graph onto a multiprocessor architecture includes three important steps — assigning actors to processors (*processor assignment*), ordering the actors assigned to each processor (*actor ordering*), and determining when each actor should commence execution. All of these tasks can either be performed at run-time or at compile time to give us different scheduling strategies.

In relation to the scheduling taxonomy of Lee and Ha [8], we focus in this paper on the *self-timed* strategy and the closely-related *ordered transaction* strategy. These approaches are popular and efficient for the DSP domain due

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 2001		**00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Mapping DSP Applications onto Self-timed Multiprocessors	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,Department of Electrical and Computer Engineering,Institute for Advanced Computer Studies,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**8**	

to their combination of robustness, predictability, and flexibility [14]. In self-timed scheduling, each processor executes the tasks assigned to it in a fixed order that is specified at compile time. Before executing an actor, a processor waits for the data needed by that actor to become available. Thus, processors are required to perform run-time synchronization when they communicate data. This provides robustness when the execution times of tasks are not known precisely or when they may exhibit occasional deviations from their compile-time estimates. Examples of an application graph and a corresponding self-timed schedule are illustrated in Figure 1.

The *ordered transaction* method is similar to the self-timed method, but it also adds the constraint that a linear ordering of the communication actors is determined at compile time, and enforced at run-time [15]. The linear ordering imposed is called the *transaction order* of the associated multiprocessor implementation. The transaction order, which is enforced by special hardware, obviates the need for run-time synchronization and bus arbitration, and also enhances predictability. Also, if constructed carefully, it can in general lead to a more efficient pattern of actor/communication operations compared to an equivalent self-timed implementation [6].

2 Modeling Self-timed Execution

In this section, we discuss two related graph-theoretic models, the *interprocessor communication graph* (*IPC graph*) G_{ipc} [14, 15] and the *synchronization graph* G_s [14], that are used to model the self-timed execution of a given parallel schedule for a dataflow graph. Given a self-timed multiprocessor schedule for G, we derive G_{ipc} by instantiating a vertex for each task, connecting an edge from each task to the task that succeeds it on the same processor, and adding an edge that has unit delay from the last task on each processor to the first task on the same processor. Also, for each edge (x, y) in G that connects tasks that execute on different processors, an IPC edge is instantiated in G_{ipc} from x to y. Figure 2 shows the IPC graph that corresponds to the application graph and self-timed schedule of Figure 1.

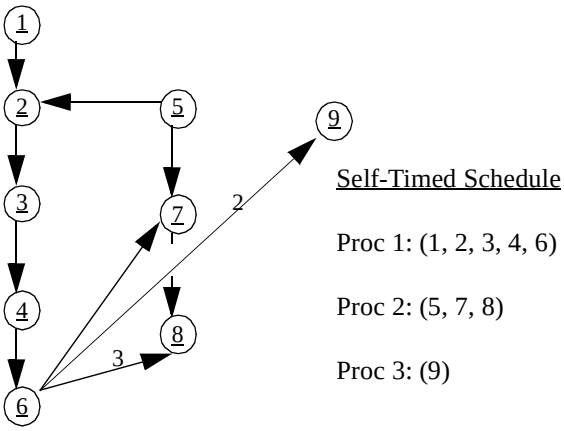

Figure 1. An example of an application graph and an associated self-timed schedule. The numbers on edges $(6, 8)$ and $(6, 9)$ denote nonzero delays.

Vertices in these graphs correspond to individual tasks of the application being implemented. Each edge in G_{ipc} and G_s is either an *intraprocessor edge* or an *interprocessor edge*. Intraprocessor edges model the ordering (specified by the given parallel schedule) of tasks assigned to the same processor. Interprocessor edges in G_{ipc}, called *IPC edges*, connect tasks assigned to distinct processors that must communicate for the purpose of data transfer, and interprocessor edges in G_s, called *synchronization edges*, connect tasks assigned to distinct processors that must communicate for synchronization purposes.

Each edge (v_j, v_i) in G_{ipc} represents the *synchronization constraint*

$$start(v_i, k) \geq end(v_j, k - delay((v_j, v_i))) \text{ for all } k, \quad (1)$$

where $start(v, k)$ and $end(v, k)$ respectively represent the time at which invocation k of actor v begins execution and completes execution, and $delay(e)$ represents the delay associated with edge e.

Initially, the synchronization graph G_s is identical to G_{ipc}. However, various transformations can be applied to G_s in order to make the overall synchronization structure more efficient. After all transformations on G_s are complete, G_s and G_{ipc} can be used to map the given parallel schedule into an implementation on the target architecture. The IPC edges in G_{ipc} represent buffer activity, and are implemented as buffers in shared memory, whereas the synchronization edges of G_s represent synchronization constraints, and are implemented by updating and testing flags in shared memory. If there is an IPC edge as well as a synchronization edge between the same pair of tasks, then a synchronization protocol is executed before the buffer corresponding to the IPC edge is accessed to ensure sender-receiver synchronization. On the other hand, if there is an IPC edge between two tasks in the IPC graph, but there is no synchronization edge between the two, then no synchronization needs to be done before accessing the shared buffer. If there is a synchronization edge between two tasks but no IPC edge, then no shared buffer is allocated between the two tasks; only the corresponding synchronization protocol is invoked.

Any transformation that we perform on the synchronization graph must respect the synchronization constraints im-

plied by G_{ipc}. If we ensure this, then we only need to implement the synchronization edges of the optimized synchronization graph. If $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are synchronization graphs with the same vertex-set and the same set of intraprocessor edges (edges that are not synchronization edges), we say that G_1 ***preserves*** G_2 if for all $e \in E_2$ such that $e \notin E_1$, we have

$$\rho_{G_1}(src(e), snk(e)) \leq delay(e), \qquad (2)$$

where $src(e)$ ($snk(e)$) represents the actor at the source (sink) of edge e; and $\rho_G(x, y) \equiv \infty$ if there is no path from x to y in the synchronization graph G, and if there is a path from x to y, then $\rho_G(x, y)$ is the minimum over all paths p directed from x to y of the sum of the edge delays on p. The following theorem (developed in [14]) is fundamental to synchronization graph analysis.

Theorem 1: The synchronization constraints (from (1)) of G_1 imply the constraints of G_2 if G_1 preserves G_2.

Theorem 1 underlies the validity of a variety of useful synchronization graph transformations, which include systematic removal of redundant synchronization edges; rearrangement of synchronization edges to trade-off latency and throughput; graph transformations for use of low-overhead synchronization protocols; and streamlined sizing of interprocessor communication buffers [14].

3 Ordering Communication

The IPC graph is an instance of Reiter's *computation graph* model [11], also known as the *timed marked graph* model in Petri net theory [10], and from the theory of such graphs, it is well known that in the ideal case of unlimited bus bandwidth, the average iteration period for the ASAP execution of an IPC graph is given by the *maximum cycle mean* (*MCM*) of G_{ipc}, which is defined by

$$MCM(G_{ipc}) = \max_{\text{cycle C in } G_{ipc}} \left\{ \frac{\sum_{v \in C} t(v)}{Delay(C)} \right\} . \qquad (3)$$

The maximum cycle mean is thus the maximum over all directed cycles C of the sum of the task execution times in C divided by the sum of the edge delays in C. The quotient in (3) is referred to as the *cycle mean* of the associated cycle C. A variety of efficient, low polynomial-time algorithms have been developed for computing maximum cycle means (e.g., see [5]).

IPC costs (estimated transmission latencies through the multiprocessor network) can be incorporated into the IPC graph model, and the performance expression (3), by explicitly including *communication* (*send* and *receive*) *actors*, and setting the execution times of these actors to equal the associated IPC costs. In this case, the performance estimate (3) is limited by any underlying uncertainties in the actor execution times, and run-time contention due to shared communication resources. Nevertheless, it has proven to be a useful estimate of performance during design space exploration for multiprocessor DSP.

A similar data structure, which is useful in analyzing ordered transaction implementations, is Sriram's *ordered transaction graph* model [15]. Given an ordering $O = \{o_1, o_2, \ldots o_p\}$ for the communication actors in an IPC graph $G_{ipc} = (V_{ipc}, E_{ipc})$, the corresponding ordered transaction graph $\Gamma(G_{ipc}, O)$ is defined as the directed graph $G_{OT} = (V_{OT}, E_{OT})$, where

$$V_{OT} = V_{ipc},\ E_{OT} = E_{ipc} \cup E_O, \qquad (4)$$

$$E_O = \{(o_p,o_1), (o_1,o_2), (o_2,o_3), \ldots, (o_{p-1},o_p)\}, \qquad (5)$$

$$delay(o_i, o_{i+1}) = 0 \text{ for } 1 \leq i < p, \text{ and } delay(o_p, o_1) = 1. \qquad (6)$$

Thus, an IPC graph can be modified by adding edges obtained from the ordering O to create the ordered transaction graph.

A closely related data structure is the *transaction partial order graph* G_{TPO} that is computed from the IPC graph by first deleting all edges in G_{ipc} that have delays of one or more, and then deleting all of the computation actors. The transaction partial order graph represents the minimum set of dependencies imposed among different processors by the communication actors of the IPC graph. These dependencies must be obeyed by any ordering of the communication operations.

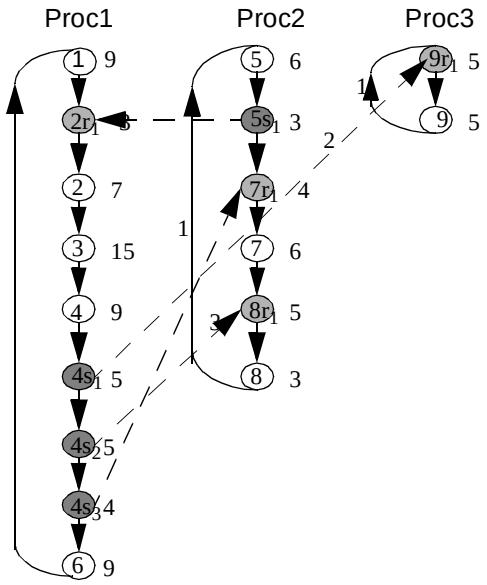

Figure 2. The IPC graph constructed from the application graph and schedule of Figure 1.

Figure 4 shows an example of a transaction partial order graph.

As described in Section 1, when the ordered transaction strategy is implemented using a hardware method such as a micro-controller that imposes the linear order, there is no need for synchronization and contention is also eliminated. Therefore, if the execution time estimates for the actors are accurate or are true worst-case values, then the MCM of the ordered transaction graph gives us an accurate estimate or worst-case bound, respectively, of the iteration period of the associated application graph under the ordered transaction strategy. Such efficient, accurate performance assessment is useful for design space exploration in general, and it is especially useful when implementing applications that have real time constraints.

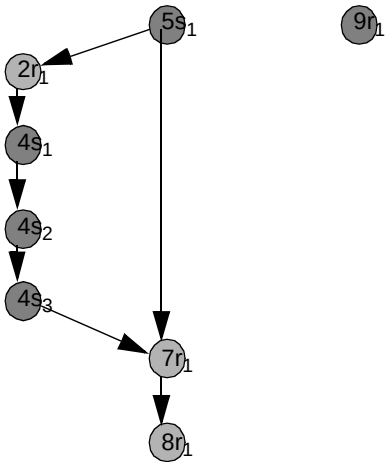

Figure 3. The transaction partial order graph constructed from IPC graph of Figure 2.

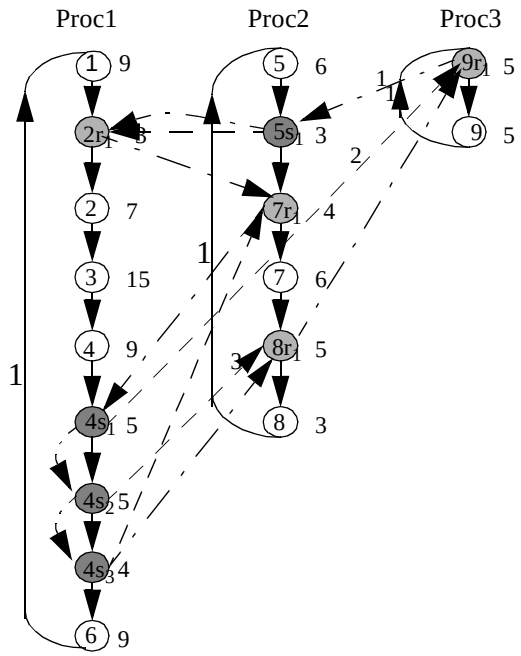

Figure 4. An example of an ordered transaction graph that is derived by the transaction partial order algorithm.

If interprocessor communication costs are negligible, an optimal transaction order can be computed in low polynomial time for a given self-timed schedule [15]. We call this method of deriving transaction orders the *Bellman Ford Based* (*BFB*) method since it is based on applying the Bellman-Ford shortest path algorithm to an intermediate graph that is derived from the given self-timed schedule.

However, when IPC costs are not negligible, as is frequently and increasingly the case in practice, the problem of determining an optimal transaction order is NP-hard [6]. Thus, under nonzero IPC costs, we must resort to heuristics for efficient solutions. Furthermore, the polynomial-time BFB algorithm is no longer optimal, and alternative techniques that account for IPC costs are preferable.

In the presence of non-negligible communication costs, an efficient transaction order can be constructed with the help of the transaction partial order graph G_{TPO} described earlier. The *transaction partial order algorithm* proceeds by considering — one by one — each vertex of G_{TPO} that has no input edges (vertices in the transaction partial order graph that have no input edges are called *ready* vertices) as a *candidate* to be scheduled next in the transaction order. Interprocessor edges are inserted from each candidate vertex to all other ready vertices in G_{ipc}, and the corresponding MCM is measured. The candidate whose corresponding MCM is the least when evaluated in this fashion is chosen as the next vertex in the ordered transaction, and deleted from G_{TPO}. This process is repeated until all communication actors have been scheduled into a linear ordering.

Figure 4 shows an example of an ordered transaction graph that is derived using the transaction partial order algorithm.

While the ordered transaction method is useful in its total elimination of run-time synchronization and arbitration overhead, the transaction partial order heuristic is able to improve the performance beyond what is achievable by a self-timed schedule even if synchronization and arbitration costs are negligible compared to actor execution times [6]. Such performance benefit is achieved by strategic positioning of the communication operations in ways that do not evolve from the natural evolution of self-timed schedules.

4 The Period Graph Model

Recall that given predictable actor execution times, one can apply (3) to accurately assess system throughput in the absence of any contention for communication resources. However, with the use of shared buses, which are employed in many embedded multiprocessor architectures, the accu-

racy of estimates based on (3) can be expected to degrade with the level of bus contention that results at run-time. Fortunately, this does not affect the validity or utility of the communication and synchronization management techniques discussed in section 2, since these techniques operate directly on the sets of interprocessor communication and synchronization edges, without need for performance estimation. Furthermore, this limitation is not encountered when using the ordered transaction model (Section 3) since contention is eliminated under this implementation model regardless of the medium used for communication.

However, accurate performance assessment of self-timed systems involving shared communication resources in general must be able to handle contention on these resources. One consequence of this contention is that under iterative execution that is self-timed, there is no known method for deriving an analytical expression for the throughput of the system, and thus, simulation is required to get a clear picture of application performance. However, simulation is computationally expensive, and it is highly undesirable to perform simulation inside the innermost optimization loop during synthesis.

The *period graph* is an efficient estimator for the system throughput that can be employed to avoid such inner-loop simulation [2]. In particular, the reciprocal of the MCM of the period graph can be used as an efficient estimate of the throughput.

If communication resource contention is resolved deterministically, and execution times are constant, then self-timed evolution may lead to an initial transient state, but the execution will eventually become periodic [14]. This holds because the multiprocessor may be modeled as a finite-state system, and thus, aperiodic behavior — which implies the presence of infinitely many distinct states — cannot hold. In DSP systems, although execution times are not always constant, or known precisely, they typically adhere closely to their respective estimates with high frequency. Under such conditions, the periodic execution pattern obtained from the estimated execution times provides an estimate of overall system throughput based on the task-level estimates. Due to the largely deterministic nature of DSP applications, such system-level performance analysis, and optimization based on task-level estimates is common practice in the DSP design community [8].

For self-timed systems, when we apply execution time estimates to assess overall throughput, it is necessary to simulate (using the execution time estimates) past the transient state until a periodic execution pattern (steady state) emerges. Unfortunately, the duration of the transient may be exponential in the size of the application specification [14], and this makes simulation-intensive, iterative synthesis approaches highly unattractive.

The period graph model greatly reduces the rate at which simulation must be carried out during iterative synthesis. Given an assignment ν of task execution times, and a self-timed schedule, the associated period graph is constructed from the periodic, steady-state pattern of the resulting simulation. The maximum cycle mean (*MCM*) of the period graph (with certain adjustments) is then used as a computationally-efficient means of estimating the iteration period (the reciprocal of the throughput) as changes are explored within a neighborhood of ν.

The first step in the construction of the period graph is the identification of the period from the simulator output. This can be performed by tracing backward through the simulation and searching for the latest intermediate time instant t_a at which the *system state* $S(t_a)$ equals the state $S(t_f)$ obtained at the end of the simulation (here, t_f denotes the simulation time limit). If no match is found, then the end of the first period exceeds t_f, and thus, the simulation needs to be extended beyond t_f. Otherwise, the region in the simulation profile (Gantt chart) that spans the interval $[t_a, t_f]$ constitutes a (minimal) period of the simulated steady state.

Here, the system state $S(t)$ contains the execution state of each processor, which is either "idle" or is represented by an ordered pair (A, τ), where A is the task being executed at time t, and τ denotes the time remaining until the current invocation of A is completed. The state $S(t)$ also contains the current buffer sizes of all IPC buffers, as well as any information (e.g., request queue status) that is used by the protocol for resolution of communication contention. Further details on period graph extraction are developed in [1].

Figure 5(a) and Figure 5(b) illustrate an *application graph* (a dataflow specification of an application) along with a self-timed schedule; Figure 5(c) shows the periodic steady state that results from the schedule of Figure 5(a) and the execution time estimates shown in Figure 5(b); and Figure 5(d) shows the resulting period graph. The nodes in Figure 5(d) that contain diagonal stripes correspond to idle time ranges in the period, and solid black circles on edges represent delays, which model inter-iteration dependencies. Note that the steady state period may span multiple graph iterations (2 in this example), and in the period graph, this translates to multiple instances of each application graph task.

For clarity in this illustration, we have assumed negligible latency associated with IPC. As described below, non-negligible IPC costs can easily be accommodated in the period graph model by introducing *send* and *receive* tasks at appropriate points.

As illustrated in Figure 5, the period graph consists of all the tasks comprising the period that was detected, with the idle time ranges between tasks (including those that are caused by communication contention) *also treated as nodes in the graph*. The nodes are connected by edges in the order

that they appear in the period. An edge is placed from the last node in the period for each processor to the first node in the period. This edge is given a delay value of one (to model the associated transition between period iterations), while all of the other intraprocessor edges have delay values of zero. This is done for all the processors in the system. Our model utilizes *send* and *receive* nodes for IPC. For each IPC point, a send node is placed on the processor that is sending data, and a corresponding receive node is placed on the processor that will receive the data. The period graph is completed by adding an edge from each send node to its corresponding receive node.

Once the period graph has been constructed, it can be used as an efficient estimator for the throughput in any optimization for which the execution times of the nodes are varied (e.g., when exploring migrations between hardware and software, applying voltage scaling, or exploring alternative processor assignments in a heterogeneous multiprocessor). However, it is not obvious how one should adjust the idle times in the period graph.

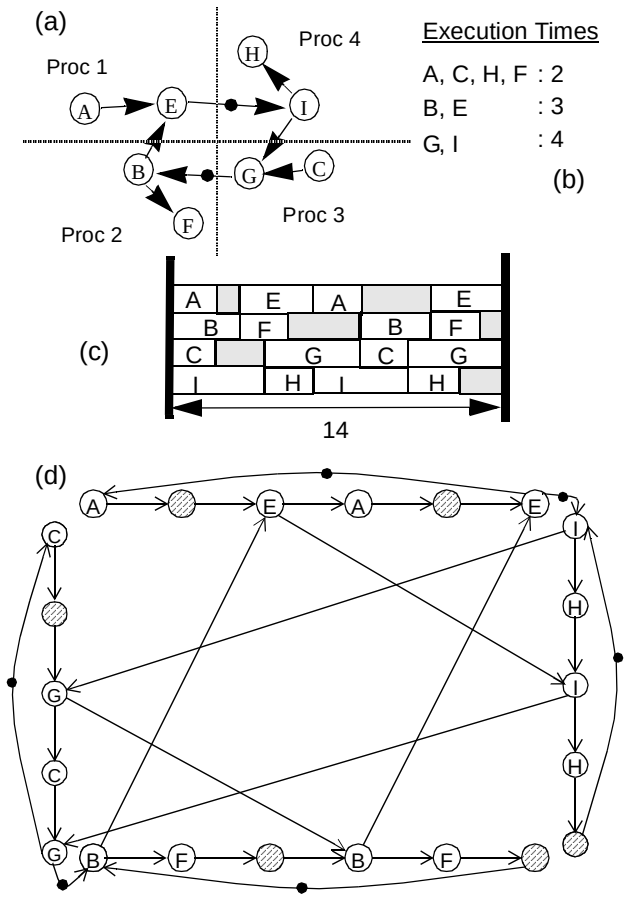

Figure 5. An illustration of the period graph model.

For this purpose, it is useful to separate the idle nodes into two sets. When a node has the necessary data to execute, but is idle waiting for access to the bus, the associated idle node is classified as a *contention idle*. When a node is idle waiting for its predecessors' data, the associated idle node is classified as a *data idle*. The effects of contention can be captured efficiently with high estimation accuracy by ignoring (setting to zero) the data idles and leaving the contention idles constant as the computation times are scaled [2].

The period graph has been applied to the problem of voltage scaling for power reduction of multiprocessor DSP systems. It has been shown to increase overall power optimization efficiency significantly when used to explore voltage variations within a limited range around a given voltage vector (assignment of processor voltages) [2]. For larger changes in node execution times, the fidelity (accuracy) of the estimate decreases. In general, one would use the period graph in a local search, for which the fidelity is acceptable, and re-simulate and rebuild the period graph outside this region when necessary. This integration of period graph analysis with occasional re-simulation has been studied in [3].

5 Clusterization Function Representations

Clustering is often used as a front-end to multiprocessor system synthesis tools. In this context, clustering refers to the grouping of actors into subsets that execute on the same processor. The purpose of clustering is thus to constrain the remaining steps of synthesis, especially scheduling, so that they can focus on strategic processor assignments.

In the context of embedded system implementation, one limitation shared by many existing clustering and scheduling techniques is that they have been designed for general purpose computation. In the general-purpose domain, there are many applications for which short compile time is of major concern. In such scenarios, it is highly desirable to ensure that an application can be mapped to an architecture within a matter of seconds. The internalization algorithm [12] and the dominant sequence algorithm [16] are examples of such low complexity algorithms.

Several probabilistic search approaches to multiprocessor scheduling have been proposed in the literature, such as *genetic algorithms*, that exploit the increased compile time tolerance available with embedded systems (e.g., see [1] for a general discussion of genetic algorithms, and [4] for an example of a recent genetic algorithm approach to scheduling). However, these approaches typically have complex solution representations in the underlying genetic algorithm formulation, and require "repair" mechanisms that further reduce their search efficiency.

The *clusterization function representation* is a mechanism for encoding candidate clustering solutions that is amenable to probabilistic search strategies, perhaps most notably to genetic algorithms, but that avoids the asymmetries and repair requirements that plague the effectiveness of conventional solution encodings that are used during scheduling [7]. The clusterization function concept is captured by the following definition.

Definition 1: Suppose that β is a subset of task graph edges. Then $f_\beta : E \rightarrow \{0,1\}$ denotes the *clusterization function* associated with β. This function is defined by:

$$f(e_i) = \begin{cases} 0 \text{ if } (e_i \in \beta) \\ 1 \text{ otherwise} \end{cases}, \quad (7)$$

where E is the set of communication edges and e_i denotes the i th edge of this set.

When using a clusterization function to represent a clustering solution, the edge subset β is taken to be the set of edges that are contained in clusters. An illustration is shown in Figure 6.

This subset view of clustering develops a natural and efficient mappings into the framework of genetic algorithms. Derived from the *schema* theory (a schema denotes a similarity template that represents a subset of $\{0, 1\}^n$), canonical genetic algorithms (which use binary representation of solution spaces) provide near-optimal sampling strategies. Furthermore, binary encodings in which the semantic interpretations of different bit positions exhibit high symmetry (e.g., with the clusterization function, each bit corresponds to the existence or absence of an edge within a cluster) allow search techniques to leverage extensive prior research on genetic operators for symmetric encodings rather than forcing the development of specialized, less-thoroughly-tested operators to handle the underlying non symmetric representation. Accordingly, the clusterization function encoding scheme is favored both by schema theory, and significant prior work on genetic operators. Furthermore, by providing no constraints on genetic operators, clusterization functions preserve the natural behavior of genetic algorithms. Finally, a clusterization function encoding never generates an illegal or invalid solution, and thus saves repair-related synthesis time that would otherwise have been wasted in locating, removing or correcting invalid solutions.

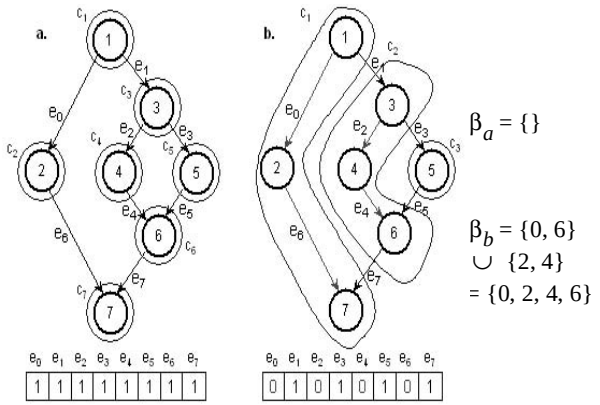

Figure 6. (a) Original task graph and corresponding clusterization function f_β; (b) a clustering of the task graph and the resulting clusterization function; (c) the associated subsets β of "zeroed edges" in this clustering.

The clusterization function approach has been applied to develop a genetic algorithm that schedules DFGs to minimize the latency of each DFG iteration (makespan). In this approach, the initial genetic algorithm population is initialized with a random selection of clusterization functions (mappings from E into $\{0, 1\}$) and the fitness is evaluated using a modified version of list scheduling that abandons the restrictions imposed by a global scheduling clock, as proposed in [13]. This application of the clusterization function has been shown to significantly outperform existing clustering techniques, including the internalization algorithm, the dominant sequence algorithm, and randomized versions of the internalization and dominant sequence algorithms that were evaluated under equal amounts of synthesis time (equal amounts of time available for probabilistic search) [7].

Since clustering is widely applicable as a front-end to many multiprocessor design contexts, and the CFA formulation captures all possible clustering alternatives in an efficient and elegant representation, it is suitable for use in many types of tools for DSP system synthesis.

6 Summary

Designers of co-design and system synthesis tools for DSP can exploit the use of predictable, coarse-grain programming models, such as synchronous dataflow (SDF), which are considered too restrictive for general-purpose design tools. However, at the same time, multiprocessor DSP implementation is typically faced with an unusually complex range of design constraints and objectives. To help address this increasing tend toward high design complexity, this paper has discussed several SDF-based intermediate representations for self-timed implementation of multiprocessor DSP applications, including the interprocessor communication graph for modeling the placement of IPC operations; the synchronization graph for separating synchronization from data transfer during IPC; the ordered transaction and transaction partial order graphs for modeling and optimizing linear orderings of communication operations; the period graph for accurate design space exploration under communication resource contention; and the clusterization function concept for representing processor assignments during the scheduling process.

Acknowledgements

Portions of this research were sponsored by the U. S. National Science Foundation (9734275), the U. S. Army Research Laboratory (under Contract No. DAAL01-98-K-0075 and the MICRA program), and the Defense Advanced Research Projects Agency (MDA972-00-1-0023, through Brown University).

References

[1] T. Back, U. Hammel, and H-P Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3-17, April 1997.

[2] N. K. Bambha and S. S. Bhattacharyya, "A Joint Power/Performance Optimization Technique for Multiprocessor Systems Using a Period Graph Construct," *Proceedings of the International Symposium on Systems Synthesis*, pages 91-97, September 2000.

[3] N. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Hybrid search strategies for dynamic voltage scaling in embedded multiprocessors. In *Proceedings of the International Workshop on Hardware/Software Co-Design*, pages 243-248, Copenhagen, Denmark, April 2001.

[4] R.C. Correa, A. Ferreira, P. Rebreyend, "Scheduling Multiprocessor Tasks with Genetic Algorithms," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 0, 825-837, 1999.

[5] A. Dasdan and R. K. Gupta, "Faster Maximum and Minimum Mean Cycle Algorithms for System-Performance Analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889-899, October 1998.

[6] M. Khandelia and S. S. Bhattacharyya. Contention-conscious transaction ordering in embedded multiprocessors. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 276-285, Boston, Massachusetts, July 2000.

[7] V. Kianzad and S. S. Bhattacharyya. Multiprocessor clustering for embedded systems. In *Proceedings of the European Conference on Parallel Computing*, pages 697-701, Manchester, United Kingdom, August 2001.

[8] E. A. Lee and S. Ha, "Scheduling Strategies for Multiprocessor Real Time DSP," *Proceedings of the Global Telecommunications Conference*, November 1989.

[9] E.A. Lee, D.G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, February, 1987.

[10] J. L. Peterson, *Petri Net Theory and Modeling of Systems*, Prentice-Hall Inc., Englewoods Cliffs, New Jersey, 1981.

[11] R. Reiter, "Scheduling Parallel Computations," *Journal of the Association for Computing Machinery*, Vol. 15, No. 4, pp. 590-599, Oct. 1968.

[12] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

[13] G. C. Sih, E. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, 1993.

[14] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[15] S. Sriram and E. A. Lee, "Determining the Order of Processor Transactions in Statically Scheduled Multiprocessors," *Journal of VLSI Signal Processing*, March, 1997.

[16] T.Yang, A.Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 951-967, 1994.